Exploring the Effects of Filter Size and Stride in CNNs

Introduction

Convolutional Neural Networks (CNNs) have become a cornerstone of modern machine learning, particularly in the realm of image processing and computer vision. These networks have powered advancements in areas as diverse as medical imaging, autonomous vehicles, and facial recognition. At the heart of CNNs lies the convolution operation, a process that extracts meaningful features from input data. This operation is controlled by two critical hyperparameters: **filter size** and **stride**.

- **Filter size** determines the scope of patterns the network can identify—think of it as the zoom level of a camera. Smaller filters focus on fine-grained details, such as edges or textures, while larger filters capture broader patterns like shapes or regions.
- **Stride**, on the other hand, controls the step size of the filter as it moves across the input. Larger strides reduce computational complexity by skipping more pixels, but this comes at the cost of resolution in the feature map.

To understand their impact, imagine observing the world through a series of camera lenses: a wide-angle lens offers a panoramic view, while a macro lens reveals intricate details. Similarly, filter size and stride determine how a CNN perceives and processes visual data.

This tutorial will guide you through the interplay between these parameters, demonstrating their influence on feature extraction. By the end, you'll understand how to adjust filter size and stride for specific tasks.

Hands-On Coding Tutorial: Exploring Convolutional Parameters in CNNs

This hands-on section introduces the practical implementation of convolution operations with varying **filter sizes** and **strides** using Python. By following the tutorial, you will:

1. **Load a sample grayscale image**.
2. **Apply convolution with configurable filters and strides**.
3. **Visualize the resulting feature maps** to understand the impact of these parameters.
4. **Experiment with custom configurations** to solidify your understanding.

Step 1: Load and Prepare an Image

We use skimage to load a sample image. For simplicity, we resize it to 128×128 pixels, reducing computational overhead while retaining sufficient detail for our experiments.

```
[5]  import numpy as np
     import matplotlib.pyplot as plt
     from scipy.signal import convolve2d
     from skimage import data
     from skimage.transform import resize

     # Load an example grayscale image (no need to convert to grayscale)
     image = data.camera()  # Sample grayscale image from skimage
     image = resize(image, (128, 128))  # Resize to make operations faster
     plt.figure(figsize=(5, 5))
     plt.imshow(image, cmap='gray')
     plt.title("Original Image")
     plt.axis('off')
     plt.show()
```

Key Takeaway: The sample image will serve as input for our convolution operations.

Step 2: Define a Convolution Function
The convolution function uses a **kernel** (filter) to scan the image and compute the resulting feature map. We employ the convolve2d function from scipy.signal for simplicity.

```
# Define a function to apply convolution
def apply_convolution(image, filter_size, stride):
    # Create a simple filter
    kernel = np.ones((filter_size, filter_size)) / (filter_size**2)  # Averaging filter
    feature_map = convolve2d(image, kernel, mode='valid')[::stride, ::stride]
    return feature_map
```

Kernel: An averaging filter (sums pixel values within its window and averages them).
Stride: Determines how far the kernel moves in each step.

Step 3: Experiment with Different Configurations

We experiment with four configurations:

1. Filter size: 3×3, Stride: 1.
2. Filter size: 5×5, Stride: 1.
3. Filter size: 3×3, Stride: 2.
4. Filter size: 5×5, Stride: 2.

Four different configurations of filter size and stride are tested, ranging from small filters with minimal stride to larger filters with greater stride. The results, visualized using Matplotlib, show that larger filters produce smoother images due to increased averaging, while higher strides reduce the resolution by skipping rows and columns in the output. This experiment illustrates the effect of these parameters..

```
# Test convolution with varying filter sizes and strides
configurations = [
    {"filter_size": 3, "stride": 1},
    {"filter_size": 5, "stride": 1},
    {"filter_size": 3, "stride": 2},
    {"filter_size": 5, "stride": 2},
]

# Visualize the results
plt.figure(figsize=(12, 10))
for idx, config in enumerate(configurations, 1):
    filter_size = config['filter_size']
    stride = config['stride']
    feature_map = apply_convolution(image, filter_size, stride)

    plt.subplot(2, 2, idx)
    plt.imshow(feature_map, cmap='gray')
    plt.title(f"Filter Size: {filter_size}, Stride: {stride}")
    plt.axis('off')

plt.tight_layout()
plt.show()
```

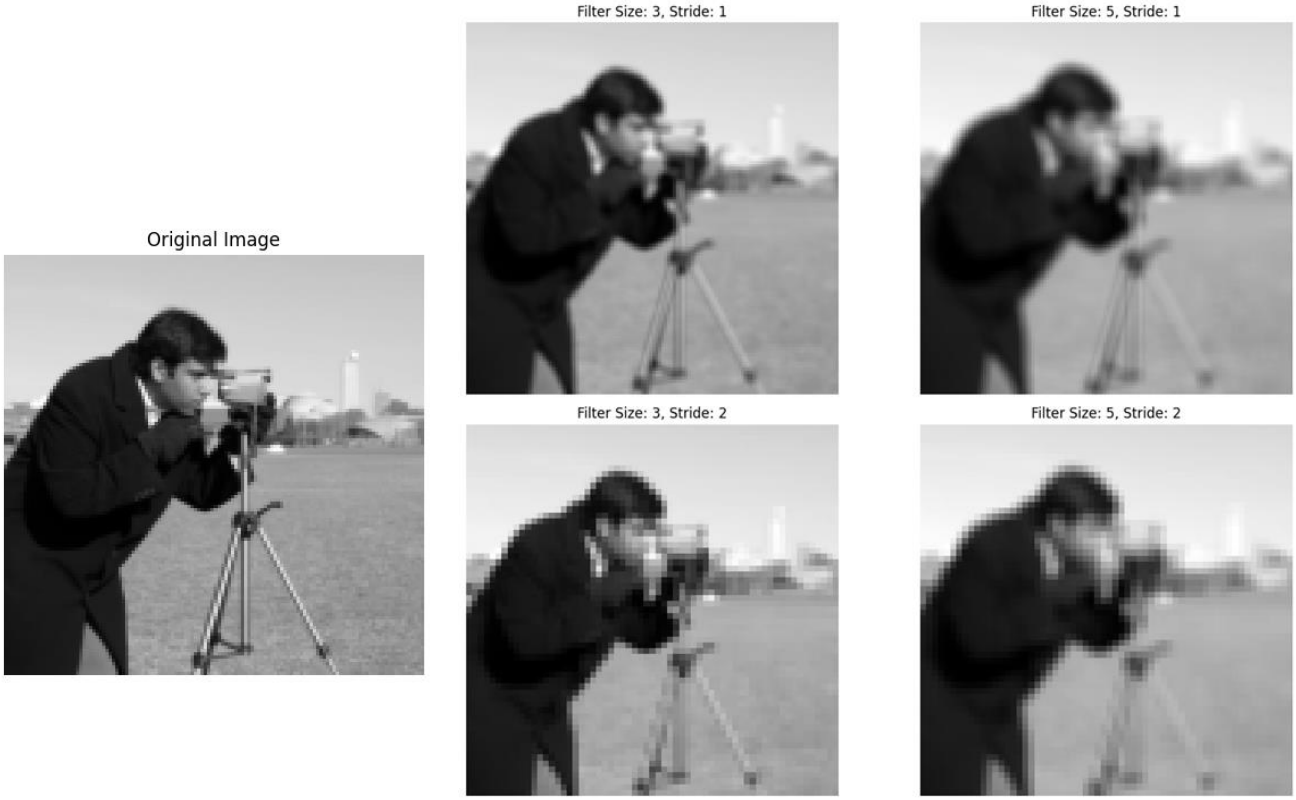
Step 4: Understanding the Results

Filter Size:

- Smaller filters (e.g., 3×3) capture finer details like edges or textures.
- Larger filters (e.g., 5×5) detect broader patterns but blur smaller details.

Stride:

- Stride of 1 provides high-resolution feature maps, preserving spatial details.
- Stride of 2 or more reduces feature map resolution, skipping finer details to emphasize larger patterns.



The Convolution Process

Convolution involves a **filter** (or kernel) moving across an input image. At each step, the filter performs an element-wise multiplication with the image patch it overlaps, and the results are summed to produce a single value in the feature map.

Filter Size: The dimensions of the stencil. A 3×3 filter looks at a 3×3 grid of pixels, while a 5×5 filter analyses a larger patch.

Stride: The step size the filter takes when moving across the image. A stride of 1 examines every possible position, while a stride of 2 skips every other position.

Effects of Filter Size

Small Filters (e.g., 3×3):
Capture fine-grained details such as edges, textures, or small patterns. However, they may miss larger features due to their limited coverage.
Example: Detecting the edge of a single leaf in a forest image.

Large Filters (e.g., 7×7):
Focus on broader patterns or structures, like large shapes or general regions. However, they may overlook small, intricate features.
Example: Recognizing a tree's silhouette in the same forest image.

Effects of Stride

Stride = 1: Produces a high-resolution feature map that preserves fine spatial details.
Example: Identifying minute changes in texture on a fabric.

Stride = 2 or Higher: Produces a low-resolution feature map, reducing computational cost but losing some spatial resolution.
Example: Spotting general contours in a large-scale aerial image.

Real-World Applications of Filter Size and Stride in Convolutional Neural Networks

The choice of **filter size** and **stride** significantly impacts the performance of Convolutional Neural Networks (CNNs) in various industries. Understanding their role allows fine-tuning of networks for optimal feature extraction and improved accuracy in specific tasks. Let's explore how these parameters are tailored to real-world applications.

Medical Imaging: Small Filters for Precise Detection

Medical imaging tasks such as detecting tumours or anomalies in X-rays, MRIs, or CT scans require high precision. Small filters (e.g., 3×3) are critical as they capture fine-grained details, including subtle changes in texture or shape that may indicate a tumour. The small stride (e.g., stride = 1) ensures no loss of spatial resolution, which is essential for accurately locating tiny abnormalities.

Example: In breast cancer detection, 3×3 filters identify microcalcifications, an early sign of cancer.

Autonomous Vehicles: Large Filters for Broader Context

Autonomous vehicles need to process their surroundings efficiently. Large filters (e.g., 7×7 or 9×9) help detect and classify large-scale objects like roads, lanes, and obstacles, ensuring safety and proper navigation. Strides greater than 1 (e.g., stride = 2) reduce computational load by downsizing the feature map, enabling faster real-time processing without losing critical context.

Example: In self-driving systems, large filters capture lane boundaries and traffic signs in one pass, providing broader scene understanding.

Satellite Imaging: Intermediate Filters for Patterns

Satellite imaging involves identifying weather patterns, vegetation, and urban sprawl. Intermediate filters (e.g., 5×5) strike a balance between capturing details and broader patterns, making them ideal for detecting cyclones, rainfall regions, or deforestation zones. Strides are adjusted (e.g., stride = 2) to ensure efficient processing of vast geographical data.

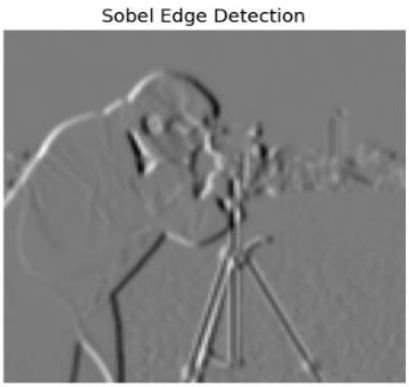
Example: 5×5 filters extract mid-scale features like cloud formations, while strides reduce computational costs over high-resolution satellite images.

Sobel filter

Replacing the default averaging filter with a custom kernel allows the convolution to specialize in extracting features such as edges, corners, or textures. For example, Sobel filters highlight edges in specific directions.

```python
kernel = np.array([[-1, 0, 1],
                   [-2, 0, 2],
                   [-1, 0, 1]])  # Horizontal Sobel filter
```

These experiments provide insight into how convolutional layers can be tuned for specific tasks, such as fine-grained detection or general pattern recognition. By customizing these parameters, you can better understand the mechanics of convolution and optimize models for diverse applications.



MNIST Dataset

The MNIST dataset consists of grayscale images of handwritten digits (0-9), each with a resolution of 28x28 pixels. The following steps were used to prepare the data:

Normalization: Pixel values were normalized to the range [-1, 1], which stabilizes training and accelerates convergence.

Data Loaders: The dataset was divided into training and testing batches (batch size = 64) to facilitate efficient model training and evaluation.

```python
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
import matplotlib.pyplot as plt

# Data preparation
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5,), (0.5,))])
train_dataset = datasets.MNIST(root='./data', train=True, download=True, transform=transform)
test_dataset = datasets.MNIST(root='./data', train=False, download=True, transform=transform)

train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=64, shuffle=False)
```

Model Architecture

We will build a CNN with two convolutional layers followed by two fully connected layers. The architecture is designed to dynamically calculate the output size after convolution based on the filter size and stride.

Key Components:

- **Conv1**: Extracts low-level features using 32 filters.
- **Conv2**: Extracts higher-level features using 64 filters.
- **Dynamic Flattening**: Adapts to varying input sizes for flexible model design.

```python
class CNNModel(nn.Module):
    def __init__(self, kernel_size, stride, input_size=28):
        super(CNNModel, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=kernel_size, stride=stride)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=kernel_size, stride=stride)

        # Calculate flattened size dynamically
        self.flattened_size = self._get_flattened_size(input_size, kernel_size, stride)

        self.fc1 = nn.Linear(self.flattened_size, 128)
        self.fc2 = nn.Linear(128, 10)

    def _get_flattened_size(self, input_size, kernel_size, stride):
        # Helper to calculate the size of the flattened feature maps
        conv1_output_size = (input_size - kernel_size) // stride + 1
        conv2_output_size = (conv1_output_size - kernel_size) // stride + 1
        return 64 * (conv2_output_size ** 2)  # 64 channels output from conv2

    def forward(self, x):
        x = torch.relu(self.conv1(x))
        x = torch.relu(self.conv2(x))
        x = torch.flatten(x, 1)
        x = torch.relu(self.fc1(x))
        return self.fc2(x)
```

Training the Model

We will use CrossEntropyLoss and the Adam optimizer to train the CNN for 5 epochs.

```python
model = CNNModel(kernel_size=3, stride=1)

# Define the loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Train the model
train_model(model, train_loader, criterion, optimizer, num_epochs=5)
```

Visualizing Feature Maps

Feature maps from the first convolutional layer reveal the patterns learned by the CNN. By visualizing these maps, we can observe how the filter size and stride influence feature extraction.

Steps:

1. Select a single image from the test set.
2. Pass the image through the first convolutional layer.
3. Plot the resulting feature maps for analysis.

This demonstrates the effects of filter size and stride on CNN performance. Smaller filter sizes and strides yield detailed feature maps, ideal for capturing intricate patterns. Larger configurations offer computational efficiency but risk losing finer details. Understanding these trade-offs is vital for designing CNNs tailored to specific tasks. By visualizing feature maps, we gain insights into the hierarchical learning process of CNNs, enabling more informed model optimization.

Conclusion

In this tutorial, we explored the intricacies of how convolutional neural networks (CNNs) extract features using various combinations of filter sizes and strides. Here are the key takeaways:

1.Filter Size: Smaller filters (e.g., 3x3) are adept at capturing fine-grained details like edges, textures, or small patterns. In contrast, larger filters (e.g., 7x7) are better suited for identifying broader structures and abstract features, such as shapes or entire objects. The choice of filter size directly affects the level of detail in the resulting feature maps, making it crucial for task-specific tuning.

2.Stride: Stride determines how the filter moves across the input image. A stride of 1 results in overlapping receptive fields, preserving spatial detail and creating high-resolution feature maps. Larger strides (e.g., 2 or 3) lead to reduced feature map resolution and faster computation but at the cost of losing some finer details. Stride settings must balance detail preservation and computational efficiency based on application requirements.

Finally, we highlighted how customized filters, such as Sobel filters for edge detection, allow for specialized feature extraction, enhancing CNNs' versatility.

Call to Action

Now it's your turn to apply these concepts! Experiment with real-world datasets, such as medical scans or satellite imagery, and tweak the parameters to see how CNNs can be tailored to solve practical problems. Start small, observe the changes, and let your curiosity drive innovation. Understanding these foundational tools is the first step to mastering deep learning.

Two separate ipynb files are also created for you to explore and experiment, use google colab to run these files. Trying changing the filter size or the number of strides and analyse the different outputs obtained through these experiments.

References

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. Link
TensorFlow documentation. Python

GitHub Link

Machine-Learning-CNN-Tutorial

Author Name: Anton Rajeev

University of Hertfordshire

Student ID:23030678